A Module System for a Programming Language Based on the LF Logical Framework

Robert Harper Frank Pfenning

September 1992
CMU-CS-92-191

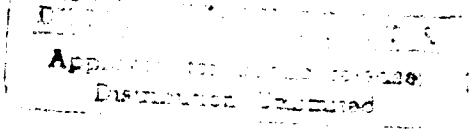
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We describe a module system for Elf, a logic programming language based on the LF logical framework. The static part of module calculus addresses name-space management and structured presentation of deductive systems. The dynamic part addresses search-space management and modularization of logic programs.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.



Keywords: logical frameworks, typed λ-calculus, dependent types, modularity, structured theories, logic programming

1 Introduction

Formal deductive systems play an important role in computer science, particularly in the areas of logic and semantics of programming languages. They are employed in three different, but obviously related, roles. Firstly, they are used to *specify* logics, type systems, operational semantics and other aspects of languages. Secondly, they form the basis for the *implementation* of such deductive systems. Thirdly, they provide an appropriate language for the formulation and proof of *meta-theorems* of programming languages.

The LF Logical Framework [11] is designed to provide an appropriate language for the high-level specification of deductive systems as they occur in logic and computer science. Its basic principle is often summarized by saying that *judgments* (the basic unit of deductive systems) are represented as *types* and *deductions* as *objects*. The framework was intentionally kept weak (by excluding, for example, polymorphism and impredicative constructs) in order to better support mechanization and to allow a simple meta-theory. This has proved auspicious: algorithms for unification have been developed [6, 26] and the type theory underlying LF has been amenable to an operational interpretation which is realized in the Elf programming language [22, 23]. Furthermore, it also seems possible to express a wide range of meta-theoretic properties of deductive systems within LF, though this line of research is only in its initial stages [17, 24, 10].

We believe that for all three tasks, *specification*, *implementation*, and *meta-theory* of deductive systems, substantial benefits can be derived from explicit structuring mechanisms for the presentation of such systems. In this paper we make a concrete proposal for a module system for the Elf programming language which attempts to address those three central issues. We have been conservative in that we only describe the part of the module calculus we understand well from the semantic and pragmatic point of view, although we do not give a formal semantics in this paper.[1] Some of the more difficult extensions we may consider in future work are mentioned in Section 6. In this paper we provide informal discussions of the meanings of various language constructs and properties. As an extended example throughout we will use two formulations of minimal propositional calculus with implication and conjunction: an axiom system in the style of Hilbert and Gentzen's calculus of natural deduction (system NJ).

The problem of modularity in the presentation of theories and logical system has been addressed from the semantical [13, 12] and the type-theoretic [4, 5, 30] point of view. Our design has been guided by these ideas and the pragmatic principles of the ML module system [16, 19]. In this paper we present at an informal level an initial design for a module system for the Elf implementation of LF [23], stressing both the static (presentational) and dynamic (search-related) aspects of the design. Many of the issues and solutions are independent of the particular core type theory chosen and we believe that the ideas presented here are also applicable to the design of module calculi for λProlog [20], Isabelle [21], or the Calculus of Constructions [3]. For further discussion of related work, the reader is referred

[1] We currently have an implementation of the static semantics of the module calculus and a separate implementation of term reconstruction and the dynamic semantics of the core language. Experience with these implementations validates the ideas in this proposal from the pragmatic point of view. We hope to have completed a full implementation of all aspects of the language by the end of 1992.

to Section 6.

The remainder of this paper is organized as follows. In Section 2 we review the LF Logical Framework as it is realized within the Elf programming language. As our approach to a module calculus is explicitly stratified (modules do not gain the status of objects, but exist in a different level of language), this core language is not modified in any essential way by the addition of modules. In Section 3 we present a calculus of *signatures* and *realizors*. Since object languages are defined by signatures, these are the centerpieces of our module calculus. Realizors provide a means of interpreting one signature into another, and hence may be used to express certain forms of "logic morphisms". In Section 4 we show how a notion of search, derived from the underlying operational semantics of Elf, can be accounted for in the module system. In Section 5 we show how limitations of realizations as defined in Section 3 can be circumvented by using relations (rather than functions) between signatures. While such relations do not have the same meta-theoretic force as realizations, they are nonetheless operationally adequate in that they can be executed to implement logic interpretations. We conclude with a brief summary of related work in Section 6 and a recapitulation of the concrete syntax in Appendix A.

2 The Core Language

We briefly review the LF logical framework [11] as realized in Elf [22, 23]. A tutorial introduction to the Elf core language can be found in [17].

The LF calculus is a three-level calculus for *objects*, *families*, and *kinds*. Families are classified by kinds, and objects are classified by *types*, that is, families of kind Type.

$$\begin{array}{lrcl} \textit{Kinds} & K & ::= & \mathrm{Type} \mid \Pi x{:}A.\ K \\ \textit{Families} & A & ::= & a \mid \Pi x{:}A_1.\ A_2 \mid \lambda x{:}A_1.\ A_2 \mid A\ M \\ \textit{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \end{array}$$

We use K to range over kinds, A, B to range over families, M, N to range over objects. a stands for constants at the level of families, and c for constants at the level of objects. In order to describe the basic judgments we consider contexts (assigning types to variables) and signatures (assigning kinds and types to constants at the level of families and objects, respectively).

$$\begin{array}{lrcl} \textit{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \\ \textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \end{array}$$

We stipulate that constants can appear only once in signatures and variables only once in contexts. This can always be achieved through renaming. $[M/x]N$ is our notation for the result of substituting M for x in N, renaming variables as necessary to avoid name clashes. We also use the customary abbreviation $A \rightarrow B$ and sometimes $B \leftarrow A$ for $\Pi x{:}A.\ B$ when x does not appear free in B. Similarly, $A \rightarrow K$ can stand for $\Pi x{:}A.\ K$ when x does not appear free in K.

The LF type theory is a formal system for deriving judgments of the following forms:

Σ *signature*	Σ *is a valid signature*
$\vdash_\Sigma \Gamma$ *context*	Γ *is a valid context*
$\Gamma \vdash_\Sigma K$ *kind*	K *is a valid kind*
$\Gamma \vdash_\Sigma A : K$	A *is a valid family of kind* K
$\Gamma \vdash_\Sigma M : A$	M *is a valid object of type* A

The derivability relation for these judgments is defined inductively by a set of inference rules. As examples, we show the rules for abstraction, application, and type-conversion at the level of objects.

$$\frac{\Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}A.\ M : \Pi x{:}A.\ B} \qquad \frac{\Gamma \vdash_\Sigma M : \Pi x{:}A.\ B \qquad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\ N : [N/x]B}$$

$$\frac{\Gamma \vdash_\Sigma M : A \qquad A \equiv A' \qquad \Gamma \vdash_\Sigma A' : \mathrm{Type}}{\Gamma \vdash_\Sigma M : A'}$$

The rule of type conversion makes use of the notion of definitional equality, which we take to be $\beta\eta$-conversion. Harper *et al.* [11] formulate definitional equality only with β-conversion and show that type checking is decidable. They also conjecture that type-checking in the system resulting from adding the η-rule would still be decidable, which has recently been proved using different techniques by Coquand [2], Salvesen [27], and Geuvers [9]. For practical purposes the formulation including the η-rule is superior, since every term has an equivalent canonical form. Thus, for us, $\equiv$ is the least congruence generated from $\beta\eta$-conversions in the usual manner.

We present concrete syntax for Elf in form of a grammar, where optional constituents are enclosed within $\langle\ \rangle$ and repeated components are shown as $cat_1 \ldots cat_n$. The concrete syntax of the core language is very closely modeled after the abstract syntax presented earlier and is also stratified. We use *term* to refer to an entity which may be from any of the three levels. In the last column we list the corresponding cases in the definition of LF above.

kindexp	::=	**type**	Type
		{*id* (:*famexp*)} *kindexp*	$\Pi x{:}A.\ K$
		famexp -> *kindexp*	$A \rightarrow K$
		(*kindexp*)	
famexp	::=	*id*	a
		{*id* (:$famexp_1$)} $famexp_2$	$\Pi x{:}A_1.\ A_2$
		[*id* (:$famexp_1$)] $famexp_2$	$\lambda x{:}A_1.\ A_2$
		famexp objexp	$A\,M$
		$famexp_1$ -> $famexp_2$	$A_1 \rightarrow A_2$
		$famexp_2$ <- $famexp_1$	$A_1 \rightarrow A_2$
		_	
		famexp : *kindexp*	
		(*famexp*)	
objexp	::=	*id*	c or x
		[*id* (:*famexp*)] *objexp*	$\lambda x{:}A.\ M$
		$objexp_1$ $objexp_2$	$M_1\,M_2$
		_	
		objexp : *famexp*	
		(*objexp*)	

The terminal *id* stands either for a bound variable, a free variable, or a constant at the level of families or objects. Bound variables and constants in Elf can be arbitrary identifiers, but free variables in a declaration or query must begin with an uppercase letter (an undeclared, unbound lowercase identifier is flagged as an undeclared constant). Identifiers may contain all characters except `(){}[]:.;%` and whitespace. `A -> B` and `B <- A` both stand for $A \rightarrow B$. The later is reminiscent of Prolog's "backwards" implication `:-` and improves the readability of some Elf programs. `->` is right associative, while the left arrow `<-` is left associative. Juxtaposition binds tighter than the arrows and is left associative. The scope of quantifications $\{x : A\}$ and abstractions $[x : A]$ extends to the next closing parenthesis, bracket, brace or to the end of the term. Term reconstruction fills in the omitted types in quantifications $\{x\}$ and abstractions $[x]$ and omitted types or objects indicated by an underscore `_`. In case of ambiguity a warning or error message results. For a description of Elf's term reconstruction phase see [23].

3 Basic Concepts

We present the basic structuring concepts of the Elf module system by a series of examples, all based on the formalization of various fragments of minimal propositional logic in LF.[2] The complete syntax of Elf appears in Appendix A.

[2]All of the examples in this and subsequent sections have been validated by our prototype implementations in the following sense. The core language implementation is used to perform term reconstruction and dynamic search, and the resulting completed forms are passed to the modules checker.

3.1 Signatures

The LF methodology of representing deductive systems consists of presenting an object language and its rules of deduction by a signature consisting of a sequence of declarations of constants together with their types or kinds. Valid types (relative to a signature) represent syntactic categories and judgments, valid objects represent abstract syntax and deductions.

The Elf module language provides a means of incrementally defining and structuring signatures. In its simplest form a signature consists of a sequence of declarations introducing constants at the level of families (`fam` declarations) and at the level of objects (`obj` declarations). At the top level a signature may be bound to a signature identifier using a `signature` binding. These features are summarized by the following grammar:

top	::=	**signature** *sigid* = *sigexp*	signature binding
	|	top_1 ; top_2	composition
sigexp	::=	**sig** *decl* **end**	encapsulation
decl	::=	**fam** *id* : *kindexp*	family
	|	**obj** *id* : *famexp*	object
	|	$decl_1$ $decl_2$	composition

It is an error to declare an identifier more than once in a signature.

To illustrate the use of the module language, we begin with the signature `MPC_LANG` of minimal propositional logic with implication, conjunction, and truth.

```
signature MPC_LANG =
sig
  fam o   : type          % Propositions

  obj => : o -> o -> o    % Implication
  obj &  : o -> o -> o    % Conjunction
  obj tt : o              % Truth
end;
```

This declaration introduces a signature identifer `MPC_LANG`, and binds it to a signature consisting of four constants, one a family constant of kind `type`, the other three object constants of the indicated types. (The symbol "`%`" begins a comment which extends to the end of the line.)

One fundamental structuring device for signatures is *inclusion*, which provides the means of splicing the declarations of one signature into those of another. We employ this device in forming the signature `HILBERT1` of Hilbert-style minimal propositional logic, defined as follows:

```
signature HILBERT1 =
sig
  % Language
  include MPC_LANG
```

```
  % Hilbert deductions
  fam |- : o -> type

  % Axioms
  obj K     : |- (=> A (=> B A))
  obj S     : |- (=> (=> A (=> B C)) (=> (=> A B) (=> A C)))
  obj ONE   : |- tt
  obj PAIR  : |- (=> A (=> B (& A B)))
  obj LEFT  : |- (=> (& A B) A)
  obj RIGHT : |- (=> (& A B) B)

  % Inference rule
  obj MP    : |- (=> A B) -> |- A -> |- B

end % signature HILBERT1
```

The declarations of `MPC_LANG` are included into `HILBERT1`, and are subsequently used in defining the deductive machinery of the Hilbert-style presentation of minimal propositional logic.

In addition to the use of `include`, the definition of the signature `HILBERT1` makes use of the implicit syntax facility of Elf [23]. Variables which are free in a `fam` or `obj` declaration are implicitly quantified within the declaration. Thus the full form of the declaration of the `K` proof constructor would be

```
obj K : {A:o} {B:o} |- (=> A (=> B A))
```

in Elf's concrete syntax. Such implicit quantifiers are tied to a form of argument synthesis (as used in LEGO [25], for example) in that the constant `K` has two implicit arguments which are determined through term reconstruction.

The prohibition on re-declaration of constants in a signature extends to the `include` declaration: an included signature may not override prior declarations, nor may subsequent declarations override included ones. We impose this restriction so as to avoid the semantic complications arising from the use of "hidden" names in ML [19]. (We may wish to reconsider this decision once we have gained more experience with the system.) To avoid possible naming conflicts, we may use a `realizor` declaration, as follows:

```
signature HILBERT2 =
sig
  % Language
  realizor L : MPC_LANG

  % Hilbert deductions
  fam |- : L.o -> type

  % Axioms
  obj K     : |- (L.=> A (L.=> B A))
```

```
  obj S     : |- (L.=> (L.=> A (L.=> B C))
                       (L.=> (L.=> A B) (L.=> A C)))
  obj ONE   : |- L.tt
  obj PAIR  : |- (L.=> A (L.=> B (L.& A B)))
  obj LEFT  : |- (L.=> (L.& A B) A)
  obj RIGHT : |- (L.=> (L.& A B) B)

  % Inference rule
  obj MP    : |- (L.=> A B) -> |- A -> |- B
end % signature HILBERT2
```

The `realizor` declaration introduces a *realizor identifier* `L` with the signature `MPC_LANG`. The effect of this declaration may be thought of as a combination of renaming and inclusion: each constant c introduced in `MPC_LANG` is renamed to `L.`c, and the resulting signature is included into the signature `HILBERT2`. This renaming applies to both declarations and uses of constants, so that `L.|-` has kind `L.o -> type`, and so on. Names of the form `L.`c are called *qualified names*, or *long identifiers*. In general a qualified name consists of a sequence of realizor identifiers, separated by periods, followed by another identifier.

The concrete syntax associated with realizor declarations and qualified names is as follows:

decl	::=	**realizor** *realid* : *sigexp*	realizor
	|	...	
longrealid	::=	⟨*longrealid* .⟩*realid*	qualified realization identifier
longid	::=	⟨*longrealid* .⟩*id*	qualified term identifier

In addition the syntax of terms is modified in the obvious fashion by allowing non-binding occurrences of *id* to be *longid*.

3.2 Realizors

In the Elf module system *realizors* provide a means of interpreting one signature (the "source") into another (the "target"). This is accomplished by providing a definition corresponding to each declaration in the target signature using the constants in the source signature. The type of the defining expression must agree with the declared type of the constant given in the target signature, taking account of the definitions of the others constants in the target on which the type may depend.

The basic forms of definition are given by the following grammar:

defn	::=	**fam** *id* ⟨: *kindexp*⟩ = *famexp*	family
	|	**obj** *id* ⟨: *famexp*⟩ = *objexp*	object
	|	$defn_1$ $defn_2$	composition

The **fam** and **obj** definitions define the identifier *id* to stand for the expression to the right of the equal sign. The optional kind or family expression may be used to constrain the

defining expression to have the ascribed kind or type; the type checker will ensure that the ascribed type or kind is correct for the defining expression. Definitions are always ***transparent*** — for purposes of type checking, a defined identifier is synonymous with its definition. Consequently, it is permissible to re-define an identifier within its scope: uses of "hidden" identifiers are simply replaced by their definitions.

Definitions may be localized using the `let` construct:

defn	::=	`let` *defn*$_1$ `in` *defn*$_2$ `end`	local definition
	|	...	
decl	::=	`let` *defn* `in` *decl* `end`	local definition
	|	...	

The semantics of a local definition is given by substitution: upon leaving the scope of a local definition, the defined identifiers are discharged by replacing their occurrences within the body of the `let` form by their definitions. For example, the definition

```
let
    obj ID = MP (MP S K) K
 in
    obj K* : |- (=> A (=> B B)) = MP K ID
end
```

is equivalent to the definition

```
obj K* : |- (=> A (=> B B)) = MP K (MP (MP S K) K)
```

Definitions may be packaged together to form a *realizor*. Realizors are introduced at top level by a *realizor binding* that specifies both the source and target signatures of the realizor. The concrete syntax of a realizor binding is as follows:

top	::=	`realizor` *realid* ⟨*parm*$_1$ ... *parm*$_n$⟩ ⟨:*sigexp*⟩ = *realexp*	realizor binding
	|	...	
parm	::=	(*decl*)	parameter declaration
realexp	::=	`real` *defn* `end`	encapsulation

The sequence of *parm*'s determines the source signature (*i.e.*, the constants that may be used within *realexp*), and the (optional) *sigexp* determines the target signature (*i.e.*, the constants that are to be defined by *realexp*). The target signature may be omitted in cases where no ambiguity arises. For example, when *realexp* is an identifier, the target signature is taken to be the (unique) signature ascribed to the identifier at the point at which it is declared or defined. When present, the target signature imposes the requirement that the given realizor *match* the given target signature, as described by example below.

When signatures are interpreted as logic definitions, realizors provide a form of "logic morphism" that allows for the expression of certain kinds of interpretations of one logical system in another. Consider the following extension of the signature `HILBERT1` given above:

```
signature HILBERT1+ =
sig
  include HILBERT1
  obj ID     : |- (=> A A)
  obj COMM_& : |- (& A B) -> |- (& B A)
end;
```

The signature HILBERT1+ presents a logical system extending that presented by the signature HILBERT1 with two additional rules of inference, expressing the reflexivity of => and the commutativity of &. Since the additional rules of HILBERT1+ are derivable within HILBERT, the extension is in fact conservative. The derivability of the additional rules may be formalized by the following realizor with source HILBERT1 and target HILBERT1+:

```
realizor Hilbert1+ (realizor H : HILBERT1) : HILBERT1+ =
real
  open H
  obj ID = MP (MP S K) K
  obj COMM_& = [x : |- (& A B)] MP (MP PAIR (MP RIGHT x))
                                   (MP LEFT x)
end;
```

The explicit *ascription* of the target signature HILBERT1+ constrains the realizor Hilbert1+ to supply definitions for all constants of HILBERT1+ using the constants of the source signature, HILBERT1. These definitions must satisfy the type constraints of HILBERT1+ as we illustrate below.

The realizor Hilbert1+ is defined using an open definition to incorporate, item by item, the constants of the signature HILBERT1. The effect of the definition open H is the same as that of including the definitions

```
  fam o = H.o
  obj => = H.=>
  % ... etc ...
  fam |- = H.|-
  obj K  = H.K
  obj S  = H.S
  % ... etc ...
```

in the body of the realizor Hilbert1+. In addition, Hilbert1+ provides explicit definitions for each of the two additional constants of the signature HILBERT1+.

The fact that Hilbert1+ is a "logic morphism" follows from the type checking obligations imposed by the presence of the result signature HILBERT1+ on the definition of the realizor Hilbert1+. Specifically, the definitions comprising the body of Hilbert1+ must be type correct in the sense that the defining expression for an identifier *id* in the signature HILBERT1+ must have the type or kind specified in HILBERT1+, after substitution of all other identifiers by their definitions. These type checking obligations include the requirements that o, which is defined to be the expression H.o, has kind type, and that =>, which is defined to be

the expression H.=>, has type H.o -> H.o -> H.o. More interesting are the type checking obligations associated with the declarations of ID and COMM_&. In particular, it must be checked that MP (MP S K) K has type |- (=> A A) bearing in mind that MP is defined to be H.MP, and so on. Since the specification of ID has implicit arguments, this requires checking that

```
(MP (=> A (=> B A)) (=> A A)
    (MP (=> A (=> (=> B A) A)) (=> (=> A (=> B A)) (=> A A))
        (S A (=> B A) A)
        (K A (=> B A)))
    (K A B))
: |- (=> A A).
```

in a context where A:o and B:o, some arguments to MP, S, and K having been determined by term reconstruction.

The type checking obligations that arise by specifying, or *ascribing*, a target signature to a realizor is called *signature matching*. We adopt two features of the ML signature matching process, namely *coercion* and *implicit instantiation*. The matching process is *coercive* in the sense that if a realizor has more components than are required by the ascribed signature, the additional definitions are winnowed out by substituting their definitions wherever they are used. The result is a "view" of the given realizor that defines precisely the identifiers declared in the ascribed signature. The matching process involves implicit instantiation when the declared type or kind of an identifier is implicitly quantified, as illustrated above. This is reminiscent of the implicit instantiation of type schemes during signature matching in ML.

Corresponding to realizor declarations we have *realizor definitions*:

defn ::= **realizor** *realid* ⟨: *sigexp*⟩ = *realexp* realizor
| ...

Realizor definitions may occur within other realizors so as to satisfy a realizor declaration in a signature. For example,

```
signature HILBERT2+ =
sig
  realizor H : HILBERT2
  let
    open H
  in
    obj ID     : |- (=> A A)
    obj COMM_& : |- (& A B) -> |- (& A B)
  end
end;

realizor Hilbert2+ (realizor H : HILBERT2) : HILBERT2+ =
real
```

```
  realizor H : HILBERT2 = H
  let
    open H
  in
    obj ID = MP (MP S K) K
    obj COMM_& = [x] MP (MP PAIR (MP RIGHT x)) (MP LEFT x)
  end
end;
```

Here we declare H to be a realizor identifier within HILBERT2+; a corresponding definition is provided in Hilbert2+.

3.3 Parameterization and Instantiation

Another important structuring device for signatures is *parameterization*, which allows us to define families of signatures indexed by objects, families, or realizors. For example, we may define the HILBERT signature parametrically in the syntax as follows:

```
signature HILBERT (realizor L : MPC_LANG) =
sig
  % Hilbert deductions
  fam |- : L.o -> type

  let  open L  in
  % Axioms
  obj K     : |- (=> A (=> B A))
  obj S     : |- (=> (=> A (=> B C)) (=> (=> A B) (=> A C)))
  obj ONE   : |- tt
  obj PAIR  : |- (=> A (=> B (& A B)))
  obj LEFT  : |- (=> (& A B) A)
  obj RIGHT : |- (=> (& A B) B)

  % Inference Rule
  obj MP    : |- (=> A B) -\ |- A -> |- B

  end  % let open L
end;  % signature HILBERT
```

Notice that we use a combination of let and open to avoid repetitive use of qualified names in the body of the signature.

Parameterized signatures may be *instantiated* by a definition matching the parameter declaration. The matching process is the same as is used to check the ascription of a target signature to a realizor: the parameter signature is the target against which the argument definition is checked. The matching process is coercive, with hidden components eliminated by substitution. Here is a simple example in which we give a parameterized form of the HILBERT+ signature. Notice that in the body we instantiate HILBERT with the parameter to

HILBERT+ so as to ensure that both are defined over the same language. In ML a sharing specification is used to the same effect, but we have chosen to use the simpler device of parameterization in our design.

```
signature HILBERT+ (realizor L : MPC_LANG) =
sig
  include HILBERT (realizor L = L)
  obj ID     : |- (L.=> A A)
  obj COMM_& : |- (L.& A B) -> |- (L.& B A)
end;
```

Realizors may also be instantiated by other realizors. An example of this is the instantiation of SYMM_&_LANG by the realizor L in the example below.

The parameterization and instantiation mechanisms of the module language are given by the following grammar:

top	::=	**signature** *sigid* $\langle parm_1 \ldots parm_n \rangle$ = *sigexp*	signature binding
	|	...	
parm	::=	(*decl*)	parameter declaration
sigexp	::=	*sigid* $\langle inst_1 \ldots inst_n \rangle$	signature instantiation
	|	...	
realexp	::=	*realid* $\langle inst_1 \ldots inst_n \rangle$	realizor instantiation
	|	...	
inst	::=	(*defn*)	parameter instantiation

We close this section with an extended example that makes the symmetry of conjunction explicit. Similar duality interpretations have been investigated and used in the IMPS system [7]. Such an interpretation is characterized by the fact that the language under consideration is interpreted in itself in a non-trivial fashion. Here, we interpret & A B as & B A.

```
realizor SYMM_&_LANG (realizor L : MPC_LANG) : MPC_LANG =
real
  fam o  = L.o
  obj => = L.=>
  obj &  = [A:o] [B:o] L.& B A
  obj tt = L.tt
end;
```

The realization which shows that this interpretation transforms theorems into theorems is non-trivial only in one case: we have to show that the translation of the PAIR axiom is a theorem. The proof we reproduce below is perhaps not the most direct Hilbert deduction of this theorem—see Section 4 how such proof objects may be constructed automatically, taking advantage of Elf's dynamic semantics. Note that type-checking guarantees that the proofs supplied below are correct.

```
realizor SYMM_&
   (realizor L : MPC_LANG)
   (realizor H : HILBERT (realizor L = L))
   : HILBERT (realizor L = SYMM_&_LANG (realizor L = L)) =
real
  let  open L  open H  in

  fam |-    = |-

  obj K     = K
  obj S     = S

  obj ONE   = ONE

  obj PAIR  : |- (=> A (=> B (& B A)))
            = MP (MP S (MP K (MP S (MP (MP S (MP K PAIR))
                                       (MP (MP S K) K)))))
                 (MP (MP S (MP K K)) (MP (MP S K) K))
  obj LEFT  : |- (=> (& B A) A) = RIGHT
  obj RIGHT : |- (=> (& B A) B) = LEFT

  % Inference Rule
  obj MP    = MP

  end  % let open
end;  % realizor SYMM_&
```

The realizor SYMM_& makes use of a *dependent signature*: the result signature is defined in terms of the parameter signature — each instance of SYMM_& determines a realizor that satisfies the corresponding instance of the signature HILBERT.

4 Search

The Elf core language has an operational interpretation which resembles Prolog's operational interpretation of Horn clauses. We will only briefly sketch it here—details and further discussion can be found in [22, 23].

Within the core language implementation, an interactive top-level similar to that of Prolog is provided in order to pose queries. A query in this context consists of a type, possibly with some free variables. This represents the goal of finding a closed object of the given type by searching through a signature in a depth-first way. As in Prolog, this employs a unification algorithm (which postpones certain equations as constraints) and back-chaining.

This only describes the search behavior to a first approximation. In order to make this operational model feasible, the programmer has a certain amount of control over how the search is performed by distinguishing *open* and *closed* families. Intuitively, a proof object

and answer substitution may contain free variables of open type, but no free variables of closed type. Put another way: a logic variable of closed type represents a goal (which is instantiated through search); a logic variable of open type on the other hand is instantiated only through unification and thus represents ordinary logic variables in the sense of Prolog. Within the module system, we prefer to call closed families *dynamic*, while we refer to open families as *static*.

We begin by defining a system of natural deduction for the minimal propositional calculus we have considered so far. We have to model the natural deduction rules reproduced below.

$$\frac{\begin{array}{c}\overline{A}^{\,x}\\ \vdots\\ B\end{array}}{A \Rightarrow B}\Rightarrow \mathrm{I}^x \qquad \frac{A \Rightarrow B \qquad A}{B}\Rightarrow \mathrm{E}$$

$$\frac{A \qquad B}{A \,\&\, B}\,\&\,\mathrm{I} \qquad \frac{A \,\&\, B}{A}\,\&\,\mathrm{E_L} \qquad \frac{A \,\&\, B}{B}\,\&\,\mathrm{E_R} \qquad \frac{}{\top}\,\top\mathrm{I}$$

The rule of implication introduction cancels all assumptions of the formula A which have been labeled with x. The transcription of these into Elf follows the standard LF methodology and is discussed in [11]. Note the implicit quantification and the representation of the deduction of B from assumption A as a function which transforms a deduction of A into a deduction of B.

```
signature NATDED (realizor L : MPC_LANG) =
sig
  let  open L  in

  % Natural deductions
  fam !    : o -> type

  % Inference rules
  obj =>I : (! A -> ! B) -> ! (=> A B)
  obj =>E : ! (=> A B) -> ! A -> ! B
  obj &I  : ! A -> ! B -> ! (& A B)
  obj &EL : ! (& A B) -> ! A
  obj &ER : ! (& A B) -> ! B
  obj ttI : ! tt

  end
end;
```

Note that this signature is parameterized over a realization of the language of minimal propositional calculus. When we relate natural deduction to the Hilbert calculus later, we can guarantee simply by instantiation that both calculi are constructed over the same language.

Next we program a very simple theorem prover for this logic. The intended operational use of this is to search for natural deductions in normal form up to a given depth bound. This does not directly construct natural deductions as they are defined above, but bounded normal deductions can be interpreted as natural deductions, as we will show later.

```
signature NAT =
sig
  fam nat : type
  obj z   : nat
  obj s   : nat -> nat
end;

signature BDD_NATDED (realizor L : MPC_LANG)
                     (realizor Nat : NAT) =
sig
  let open L
      obj s = Nat.s
  in

  % Bounded, normal deductions
  % Searching backwards from the conclusion,
  %  using only introduction rules
  fam !<  : Nat.nat -> o -> type

  % Searching forwards from the assumptions,
  %  using only elimination rules
  fam !>  : Nat.nat -> o -> type

  % Inference rules
  obj =>I<  : (({M:Nat.nat} !> (s M) A) -> !< N B)
                 -> !< (s N) (=> A B)
  obj =>E>  : !> N (=> A B) -> !< N A -> !> (s N) B
  obj &I<   : !< N A -> !< N B -> !< (s N) (& A B)
  obj &EL>  : !> N (& A B) -> !> (s N) A
  obj &ER>  : !> N (& A B) -> !> (s N) B
  obj ttI<  : !< (s N) tt
  obj close : !> N A -> !< N A

  end  % let open L ...
end;  % signature BDD_NATDED
```

Within the Elf core language implementation, the problem of finding a bounded normal deduction for the proposition $A \Rightarrow A$, where A is a propositional variable (with a rather arbitrary bound of 3), can be expressed with the following query (assuming the families !< and !> are dynamic):

```
?- {A:o} !< (s (s (s z))) (=> A A).
solved

Query = [A:o] =>I< ([p:{M:nat} !> (s M) A] close (p (s z))).
```

Within the module system, execution of a query is viewed as the process of finding a definition of an object with a declared type. There is an additional mechanism for search control in that the realizations which may be used for search have to be explicitly introduced as dynamic. Since there are no constants available at the top-level it is convenient to introduce top-level constructs which allow us to construct *initial* realizations which may be thought of as implicit parameters to subsequent realizor definitions. These come in two flavors: **dynamic** declarations are used for search, while **static** declarations are used only for type-checking. A properly scoped version to introduce dynamic declarations inside of realizors (with keyword **using**) is introduced later in this section.

top	::=	**static** *decl*	static declaration
	|	**dynamic** *decl*	dynamic declaration
	|	...	

The following top-level definitions will initiate search for a bounded normal deduction of depth at most 3:

```
static realizor L : MPC_LANG;
dynamic realizor BN : BDD_NATDED (realizor L = L);
open L;
open BN;
solve obj Query : {A:o} !< (s (s (s z))) (=> A A);
```

This will result in the top-level definition

```
obj Query = [A:o] =>I< ([p:{M:nat} !> (s M) A] close (p (s z)))
```

This example also introduces **solve**, a new form of definition.

defn	::=	**solve** *decl*	initiate search
	|	...	

When processing **solve** *decl* the interpreter tries to find definitions for all the constants declared by *decl*, using objects which are dynamically available. Operationally, logic variables will be created for all free variables and declared constants in *decl* and the Elf interpreter will then search for appropriate instances for these variables in the order of declaration. Upon success (it only looks for the first solution), the bindings of the logic variables are used to extract appropriate definitions of the declared constants by implicitly quantifying over all remaining logic variables (which must have static type). The **solve** construct cannot be applied to declarations of families: it only searches for definitions of objects.

A top-level declaration of the form **dynamic fam** *id* : *kindexp* means that the declared family will be treated as dynamic. That is, search within the scope of the declaration

(initiated by **solve**) will ensure that all free variables of dynamic type will be solved. A declaration of the form **dynamic obj** *id* : *famexp* means the the declared constant *id* will be available for backchaining search. A dynamic declaration of a realizor applies hereditarily to its constituent declarations (similarly for composition and inclusion).

To illustrate the concept of dynamic declarations and **solve**, we consider a few simple examples.

```
static fam nat : type;
static obj z : nat;
static obj s : nat -> nat;
solve obj n : nat;
```

will succeed, elaborating the last line to the definition

```
obj n = N : nat;
```

which, in full notation, would be

```
obj n = [N:nat] N : nat -> nat;
```

where the argument to n is implicit. In fact, using n anywhere subsequently will have the effect of replacing it by a free (anonymous) variable, since it will be expanded to n _, which is equal to _ by the definition of n. This is generally the case if the type of a variable to be solved is not dynamic.

```
dynamic fam nat : type;
static obj z : nat;
static obj s : nat -> nat;
solve obj n : nat;
```

This will fail, since there are no dynamic objects available to construct a term of type nat. Since nat is dynamic, no free variables of type nat are tolerated in the substitution for n. and search will fail.

```
dynamic fam nat : type;
dynamic obj z : nat;
dynamic obj s : nat -> nat;
solve obj n : nat;
```

This will succeed and bind n to z.

```
dynamic fam nat : type;
static obj z : nat;
dynamic obj s : nat -> nat;
solve obj n : nat;
```

There is no closed term of type nat which can be constructed only from dynamic objects, but the interpreter fails to recognize this, and instead loops while constructing incomplete answers of the form s (s (s ... (s N))), for logic variables N of type nat.

We now would like to make the relationship between bounded normal deductions and natural deductions explicit. Not surprisingly, this takes the form of a realization of the signature BDD_NATDED from the signature NATDED, both over the same language L. There is no particular difficulty in defining this realization: intuitively, both backwards and forwards provability judgments are interpreted as provability by simply ignoring the bounds.

```
realizor BDD_ND
   (realizor L : MPC_LANG)
   (realizor Nat : NAT)
   (realizor ND : NATDED (realizor L = L))
   : BDD_NATDED (realizor L = L) (realizor Nat = Nat) =
real
  fam !< N A = ND.! A
  fam !> N A = ND.! A

  obj =>I<  = [P] ND.=>I ([x:ND.! A] P ([n:Nat.nat] x))
  obj =>E>  = ND.=>E
  obj &I<   = ND.&I
  obj &EL>  = ND.&EL
  obj &ER>  = ND.&ER
  obj ttI<  = ND.ttI
  obj close = [P] P
end;  % realizor BDD_ND
```

Next we can combine the search for bounded normal deductions with tne interpretation above to construct an interpretation showing that natural deductions are complete with respect to Hilbert deductions. The following realization shows that all Hilbert axioms are provable within our natural deduction calculus, and that Modus Ponens can be realized as a rule of inference. The construction of this realization involves bounded search (with a bound of 5).

```
realizor Natded_Hilbert
  (realizor L : MPC_LANG)
  (realizor Nat : NAT)
  (realizor ND : NATDED (realizor L = L))
  : HILBERT (realizor L = L) =
real
  let
       obj five = Nat.s (Nat.s (Nat.s (Nat.s (Nat.s Nat.z))))
       open L
       open ND
  in
```

```
    using realizor BD = BDD_ND (realizor L = L)
                               (realizor Nat = Nat)
                               (realizor ND = ND)
    in
      solve obj K' : {A:o} {B:o} BD.!< five (=> A (=> B A))
      solve obj S' : {A:o} {B:o} {C:o}
                       BD.!< five (=> (=> A (=> B C))
                                      (=> (=> A B) (=> A C)))
    end

    obj |-     =  !
    obj K      =  K' _ _
    obj S      =  S' _ _ _
    obj ONE    =  ttI
    obj PAIR   =  =>I [x] =>I [y] &I x y
    obj LEFT   =  =>I [x] &EL x
    obj RIGHT  =  =>I [x] &ER x

    obj MP     =  =>E

  end  % let solve ...
end;  % realizor Natded_Hilbert
```

New here is the construct **using** with syntax

$$\begin{array}{rcll} \mathit{defn} & ::= & \textbf{using}\ \mathit{defn}_1\ \textbf{in}\ \mathit{defn}_2\ \textbf{end} & \text{use dynamically} \\ & | & \ldots & \end{array}$$

For each definition in defn_1 a corresponding dynamic *declaration* will be introduced. In the basic cases, this will declare the kind of a defined family, the type of a defined object, or the signature ascribed to a realizor. If the declaration can not be determined unambiguously, an appropriate error message is issued. The *definitions* of the constants in defn_1 are *not* visible for type-checking and search within defn_2. However, upon leaving the scope of the **using** statement, occurrences of the constants in defn_2 are replaced by the definition given to them in defn_1. Thus, in contrast to **let**, **using** introduces definitions which are opaque (abstract) within its scope. This yields exactly the right behavior here: while we solve for objects `K'` and `S'`, we use the signature containing `!<` and `!>` for back-chaining search. Once we have constructed the desired objects (representing bounded normal deductions) and leave the scope of **using**, the definitions are applied, interpreting the bounded normal deductions as natural deductions. Those natural deductions are then used directly to provide definitions for axioms `K` and `S` in the Hilbert calculus.

When processing **using** defn_1 **in** defn_2 **end**, realizations within defn_1 are hereditarily marked as dynamic. This is convenient in most circumstances, but it does require that declarations intended for dynamic use and those intended for static use are separated. Normally,

the definition of the syntax of an object language will be static, while signatures intended to perform search over the expressions of the object language are explicitly parameterized over a realization of the object language.

5 Signature Relations

The interpretations which can be represented using realizors within the language we have presented so far are limited by the functions which are expressible in the core language. The core language provides λ-abstraction as its only mechanism for the formation of functions. This is no accident—generalizations of the core language to admit, for example, some forms of recursion either render LF type-checking undecidable, or destroy the adequacy of encodings by admitting too many functions. Consider, for example, the rule of implication introduction in the representation of natural deductions:

$$\dfrac{\begin{array}{c}\overline{A}^{\,x}\\ \vdots\\ B\end{array}}{A \Rightarrow B} \Rightarrow \mathrm{I}^{x}$$

The premiss of this rule is represented as a function which transforms a deduction of A into a deduction of B, that is

```
obj =>I : (! A -> ! B) -> ! (=> A B)
```

In order for this to lead to an adequate encoding of natural deductions we must make sure that every object P of type `(! A -> ! B)` does in fact represent a deduction of B from the assumption A. If λ-abstraction is the only way to form functions this is guaranteed, since P must be equivalent to a term of the form `[`x `: ! A]` P', where P' is schematic in x. Under generalization, for example, to admit primitive recursion, this is no longer the case.

We call the kind of realizations which can be directly represented as a `realizor` a *uniform realization*. There are a number of reasons why certain interpretations may not be uniform. A realization may have to be defined by general or primitive recursion on the structure of an object. Or a realization may require recursion on the structure of a type. As an example, consider the following attempt to prove the deduction theorem for the Hilbert formulation of minimal propositional calculus.

```
signature DEDTHM (realizor L : MPC_LANG) =
sig
  include HILBERT (realizor L = L)
  obj dedthm : (|- A -> |- B) -> |- (=> A B)
end;
```

We cannot *uniformly* construct a deduction of `|- (=> A B)` from a deduction of `|- B` under the assumption `|- A`. Instead, we must distinguish cases, depending on the form of the deduction of `|- B`. In a hypothetical language extension by a form of primitive recursion, this might be written as following "realizor".

```
"realizor" DedThm (realizor L : MPC_LANG)
                  (realizor H : HILBERT (realizor L = L))
                  : DEDTHM (realizor L = L) =
real
  let  open L  open H  in
  obj dedthm ([x] x)      = MP (MP S K) K
    | dedthm ([x] ONE)    = MP K ONE
    | dedthm ([x] PAIR)   = MP K PAIR
    | dedthm ([x] LEFT)   = MP K LEFT
    | dedthm ([x] RIGHT)  = MP K RIGHT
    | dedthm ([x] K)      = MP K K
    | dedthm ([x] S)      = MP K S
    | dedthm ([x] MP (P x) (Q x)) = MP (MP S (dedthm P))
                                       (dedthm Q)
  end
end;
```

Does the fact that this is not a legal `realizor` mean that we cannot use the deduction theorem for the construction of deductions? Fortunately not! While we do not have a way of internally verifying through a realization that the deduction theorem is an admissible rule of inference, we can still implement the computational content of the realization above as a relation between deductions.

```
signature DEDTHM (realizor L : MPC_LANG)
                 (realizor H : HILBERT (realizor L = L)) =
sig
  let  open L  open H  in

  fam ded : (|- A -> |- B) -> |- (=> A B) -> type

  obj ded_ID    : ded ([x] x)      (MP (MP S K) K)
  obj ded_K     : ded ([x] K)      (MP K K)
  obj ded_S     : ded ([x] S)      (MP K S)
  obj ded_ONE   : ded ([x] ONE)    (MP K ONE)
  obj ded_PAIR  : ded ([x] PAIR)   (MP K PAIR)
  obj ded_LEFT  : ded ([x] LEFT)   (MP K LEFT)
  obj ded_RIGHT : ded ([x] RIGHT)  (MP K RIGHT)
  obj ded_MP    : ded ([x] MP (P x) (Q x)) (MP (MP S P') Q')
                     <- ded P P' <- ded Q Q'
  end
end;
```

We have written the final clause using the left-arrow notation in order to emphasize the operational reading of this signature. More efficient versions of this algorithm (known as bracket abstraction) can easily be implemented. For example, `ded_K` through `ded_RIGHT`

can be combined into one clause. Type-checking of the above signature guarantees that if a query of the form `solve` c : `ded` P Q succeeds, then Q will indeed be a deduction of the appropriate implication. The fact that such a query will *always* succeed whenever P is closed and Q is a free variable is easily checked by hand, but this property remains outside the type system. Some initial thoughts on the mechanization of this check are reported in [24] and applications to non-trivial problems in the theory of programming languages can be found in [17] and [10].

As a related example, consider the implementation of a translation from natural deductions into Hilbert deductions. This relation, too, defines a (non-uniform) realization and takes advantage of the deduction theorem.

```
signature HILBERT_NATDED
  (realizor L : MPC_LANG)
  (realizor H : HILBERT (realizor L = L))
  (realizor N : NATDED (realizor L = L)) =
sig
  let  open L  open H  open N  in

  realizor D  : DEDTHM (realizor L = L) (realizor H = H)

  fam ndh     : ! A -> |- A -> type

  obj ndh_=>I : ndh (=>I PP) Q
                  <- ({x:! A} {y:|- A}
                         ndh x y
                         -> ({C:o} D.ded ([z:|- C] y) (MP K y))
                         -> ndh (PP x) (PP' y))
                  <- D.ded PP' Q
  obj ndh_=>E : ndh (=>E P Q) (MP P' Q') <- ndh P P' <- ndh Q Q'
  obj ndh_&I  : ndh (&I P Q) (MP (MP PAIR P') Q')
                   <- ndh P P'
                   <- ndh Q Q'
  obj ndh_&EL : ndh (&EL P) (MP LEFT P') <- ndh P P'
  obj ndh_&ER : ndh (&ER P) (MP RIGHT P') <- ndh P P'
  obj ndh_ttI : ndh ttI ONE
  end
end;
```

Note how we used `D.ded` as an auxiliary judgment in the implication introduction rule. According to our signature specifying inference rules for natural deduction, `PP` has type `! A -> ! B` for some `A` and `B`. To translate `PP` we make the local assumption `! A` labeled `x` and then translate `PP x` (of type `! B`). The corresponding Hilbert deduction will use a corresponding hypothesis `y : |- A`. We make this correspondence explicit through the assumption `ndh x y`. This translation yields a deduction `PP' : |- A -> |- B`, but

we need to construct a deduction of `|- (=> A B)`. This is where we apply bracket abstraction (the computational content of the deduction theorem), which is done by calling `D.ded PP' Q`. It remains to explain the second local assumption in this rule, namely `{C:o} D.ded ([z:|- C] y) (MP K y)`. Recall, that bracket abstraction as implemented in the signature `DEDTHM` applies only to deductions from exactly one hypothesis. Here, however, we introduce a new, temporary hypothesis `y` which is not accounted for in the signature `DEDTHM`. Therefore we have to specify the behavior of the bracket abstraction algorithm on `y`. The obvious course of action is the same as for the axioms: simply return `MP K y`.

This example uses a declared realization within a signature instead of explicit parameterization or inclusion. The signature `HILBERT_NATDED` thus intrinsically contains the realization of the deduction theorem. As a consequence, whenever `HILBERT_NATDED` is used dynamically, the judgment `D.ded` will also be available dynamically. At the same time, unlike when `include` is employed, the name space of the realization `D` remains separate.

The computational content of relations such as the one above is sufficient to aid in the construction of other realizations. We now return to an earlier example, showing how the realization establishing the symmetry of conjunction could have been established by translating a natural deduction.

```
realizor SYMM_&'
   (realizor L : MPC_LANG)
   (realizor H : HILBERT (realizor L = L))
   (realizor N : NATDED (realizor L = L))
   (realizor HN : HILBERT_NATDED (realizor L = L)
                                 (realizor H = H)
                                 (realizor N = N))
   : HILBERT (realizor L = SYMM_&_LANG (realizor L = L)) =
real
  let  open L  open H  in

  fam |-    = |-

  obj K     = K

  obj S     = S
  obj ONE   = ONE

  using realizor HN = HN
  in
    solve obj H   : |- (=> A (=> B (& B A)))
          obj NDH : HN.ndh (N.=>I [x] N.=>I [y] (N.&I y x)) H
  end

  obj PAIR  = H

  obj LEFT  : |- (=> (& B A) A) = RIGHT
```

```
  obj RIGHT : |- (=> (& B A) B) = LEFT

  % Inference Rule
  obj MP    = MP

  end
end;  % realizor SYMM_&'
```

The two objects to be solved for, `H` and `NDH`, are solved simultaneously. Since `|-` is not dynamic, it is instantiated only through the process of solving for `NDH`, whose type is dynamic. However, we need to declare `H` here so we can use its binding as the definition for `PAIR`.

The natural deduction we give explicitly here could also have been generated through search for a bounded normal deduction and its interpretation as a natural deduction. We leave it to the reader to write out the appropriate `realizor`.

6 Related and Future Work

The design of the module system owes a great deal to the ML module system [16, 19]. We have replaced *sharing equations* by explicit parameterization, at the cost of some verbosity, but with the gain of semantic simplicity. While the similarities to the ML module system are striking in some respects, emphasis has shifted significantly. In ML, definitions within *structures* (the ML analogues of our *realizations*) give rise to computation, while in our setting of logic programming the signatures themselves are given an operational interpretation by a search procedure. Realizations only interpret the result of such computations.

The problem of modularity in logical frameworks has previously been addressed in various typed λ-calculi. De Bruijn's *telescopic mappings* [4], for example, provide for a first-class notion of contexts. Along similar lines, a type-theoretic calculus with explicit contexts called DEVA has been developed and applied to a number of interesting examples by de Groote [5] and Weber [30]. Our module calculus can be seen as a higher-level language which could be compiled into a lower-level type theory such as DEVA. The main difficulty in this compilation process is to account for constant names (which are *not* subject to α-conversion, in contrast to bound variables names) and the implicit coercions which take place when signatures or realizations are instantiated.

Much more powerful, impredicative type theories have been investigated by Luo [14] and are implemented within the LEGO system [15]. The basis for this work is the Calculus of Constructions [3] which is not intended as a logical framework, but a type theory in which constructive mathematics can be directly formalized and reasoned about internally. Explicit theory structuring is achieved through Σ-types. This provides no real name-space management — it has more the flavor of an "abstract syntax" for a module calculus. In fact, ECC provides a suitable framework in which to interpret the modularity constructs introduced here. In particular, the static semantics of our language may be given by a translation into ECC that mediates between concrete names and abstract projections associated with Σ-types. The full expressive power of ECC is not needed for this purpose, but in view of

the meta-theoretic properties of ECC (in particular, strong normalization and confluence), there is no harm in doing so.

Our thinking has also been influenced by the work of Cartmell on contextual categories [1], from which the term "realizor" is derived. The relevance of Cartmell's work stems from the central role of contexts in his treatment of generalized algebraic theories. In our setting signature expressions denote a form of context, and realizor expressions denote realizations in essentially Cartmell's sense. In particular, Cartmell's definition of realization involves precisely the form of sequential dependency that we use in our definition of signature matching. To make the connection precise would require a formal semantics in which signature expressions denote LF signatures, and in which realizor expressions denote realizations between LF signatures. This is only approximately correct, since we must also account for qualified names, nested realizor definitions, parameterized signatures, and dependent signatures. The full exposition of this semantics lies beyond the scope of this paper, but we feel confident that such an explanation may be given in terms of an extension of the LF type theory with a level of signatures and realizors. It is important to ensure that the adequacy of encodings is preserved on passage to a modular presentation; we consider this an important direction for further research.

None of the calculi mentioned above give an integral treatment to search control, which is provided within our framework. In the context of more traditional logic programming, this has been in addressed in a similar manner by Miller [18]. In Miller's approach, clauses can be added to the program through implication with its intuitionistic meaning: in order to solve the goal $D \Rightarrow G$ we assume D while solving G. In Elf this is also supported in the core language: operationally, trying to find an object of type $A \rightarrow B$ or $\Pi x{:}A.\ B$ amounts to assuming that we have an object of type A and then trying to find an object of type B. Miller then generalizes implication by allowing a module name $\mathcal{M}$ in place of the formula D. where a module is an abbreviation for a large formula. Our `using` construct serves a very similar purpose in that is makes other code available for search within a well-defined scope. However, Miller's approach does not address name-space management. It is difficult to add this, since module constructs such as $\mathcal{M} \Longrightarrow G$ can be embedded in clauses. In particular, there could be logic variables ranging over module names, which makes it impossible to statically resolve references to constants declared in external modules.

Sannella and Burstall [28] introduced a notion of modular presentation for LCF theories, with an associated search procedure that takes advantage of the structure of a theory presentation to limit the search space. These methods were generalized to an arbitrary logic, considered as an abstract family of consequence relations, by Harper, Sannella, and Tarlecki [13, 12], where their behavior under representation in a logical framework is also considered. Local declarations introduce problems of adequacy that are rectified by *ad hoc* techniques that segregate types that represent judgments from other types. Subsequently, Gardner [8] introduced a more refined notion of framework that enforces such a segregation; it seems plausible that in this setting the aforementioned problems of adequacy do not arise. The possibility of introducing local declarations in the present setting requires further investigation.

Sannella and Wallen's proposal [29] for a module system for Prolog bears certain similarities to ours as both have been inspired by ML. In their system, signatures provide

declarations of arities for predicate and function symbols and structures contain clauses defining the predicates. Thus, as in ML, structures and functors (as parameterized structures) contain code to be executed, while signatures declare what structures must define. In our approach, signatures contain declarations and code to be executed, while realizations allow us to define interpretations between signatures and thus the deductive systems they represent, adding another dimension to the module calculus.

The design presented here is deliberately conservative in that we restricted ourselves to constructs for which a firm type-theoretic basis can be established. There are a number of additional features for which this is less clear, and which we have therefore been omitted at present. We will reconsider them in the light of practical experience with a prototype implementation. The principal features we have in mind are sharing equations, higher-order realizations, local declarations, inclusion of individual declarations, and declaration of infix operators. Finally, we are considering whether the language of realizations could be strengthened by some form of primitive recursion in order to allow non-uniform realizations without destroying the basic properties of the LF type theory. We feel that the natural separation of the module level from the object level provides us with an opportunity for such an extension which would not be available in the core calculus, the LF type theory. This issue is closely related to the question of how to include inductive types in the module calculus, if the core language were generalized to a more expressive type theory such as the Calculus of Constructions.

A Syntax

Identifiers

id			term identifier
sigid			signature identifier
realid			realizations identifier
longrealid	::=	⟨*longrealid* .⟩*realid*	qualified realization identifier
longid	::=	⟨*longrealid* .⟩*id*	qualified term identifier

Core Language

kindexp	::=	**type**	Type
	|	{*id* ⟨:*famexp*⟩} *kindexp*	$\Pi x{:}A.\ K$
	|	*famexp* -> *kindexp*	$A \to K$
	|	(*kindexp*)	
famexp	::=	*longid*	a
	|	{*id* ⟨:$famexp_1$⟩} $famexp_2$	$\Pi x{:}A_1.\ A_2$
	|	[*id* ⟨:$famexp_1$⟩] $famexp_2$	$\lambda x{:}A_1.\ A_2$
	|	*famexp objexp*	$A\,M$
	|	$famexp_1$ -> $famexp_2$	$A_1 \to A_2$
	|	$famexp_2$ <- $famexp_1$	$A_1 \to A_2$
	|	_	
	|	*famexp* : *kindexp*	
	|	(*famexp*)	
objexp	::=	*longid*	c or x
	|	[*id* ⟨:*famexp*⟩] *objexp*	$\lambda x{:}A.\ M$
	|	$objexp_1$ $objexp_2$	$M_1\,M_2$
	|	_	
	|	*objexp* : *famexp*	
	|	(*objexp*)	

Declarations and Signatures

sigexp	::=	**sig** *decl* **end**	encapsulation
	|	*sigid* $\langle inst_1 \ldots inst_n \rangle$	signature instantiation
parm	::=	(*decl*)	parameter declaration
decl	::=	**fam** *id* : *kindexp*	family
	|	**obj** *id* : *famexp*	object
	|	**realizor** *realid* : *sigexp*	realizor
	|	$decl_1$ $decl_2$	composition
	|	**let** *defn* **in** *decl* **end**	local definition
	|	**include** *sigexp*	inclusion

Definitions and Realizations

realexp	::=	**real** *defn* **end**	encapsulation
	|	*longrealid* $\langle inst_1 \ldots inst_n \rangle$	instance
inst	::=	(*defn*)	parameter instantiation
defn	::=	**fam** *id* $\langle$: *kindexp* $\rangle$ = *famexp*	family
	|	**obj** *id* $\langle$: *famexp* $\rangle$ = *objexp*	object
	|	**realizor** *realid* $\langle$: *sigexp* $\rangle$ = *realexp*	realizor
	|	$defn_1$ $defn_2$	composition
	|	**let** $defn_1$ **in** $defn_2$ **end**	local definition
	|	**open** *longrealid* $\langle$: *sigexp* $\rangle$	inclusion
	|	**using** $defn_1$ **in** $defn_2$ **end**	use dynamically
	|	**solve** *decl*	initiate search

Top-Level

top	::=	**signature** *sigid* $\langle parm_1 \ldots parm_n \rangle$ = *sigexp*	signature binding
	|	**realizor** *realid* $\langle parm_1 \ldots parm_n \rangle$ $\langle$: *sigexp* $\rangle$ = *realexp*	realizor binding
	|	**static** *decl*	static declaration
	|	**dynamic** *decl*	dynamic declaration
	|	*defn*	pervasive definition
	|	top_1 ; top_2	composition

References

[1] John Cartmell. Generalized algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[2] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[3] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[4] N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91:189–204, 1991.

[5] Philippe de Groote. Nederpelt's calculus extended with a notion of context as a logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 69–86. Cambridge University Press, 1991.

[6] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.

[7] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 653–654. Springer-Verlag LNAI 449, 1990.

[8] Philippa Gardner. *Representing Logics in Type Theory.* PhD thesis, University of Edinburgh, July 1992. Available as Technical Report CST-93-92.

[9] Herman Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In A. Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 453–460, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[10] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[11] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 199? To appear. Available as Technical Report CMU-CS-89-173, Carnegie Mellon University. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[12] Robert Harper, Donald Sannella, and Andrzej Tarlecki. Logic representation. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, and A. Poigneé, editors, *Proceedings of the Workshop on Category Theory and Computer Science*, pages 250–272, Manchester, UK, September 1989. Springer-Verlag LNCS 389.

[13] Robert Harper, Donald Sannella, and Andrzej Tarlecki. Structure and representation in LF. In *Fourth Annual Symposium on Logic in Computer Science*, pages 226–237, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[14] Zhaohui Luo. ECC, an extended Calculus of Constructions. In *Fourth Annual Symposium on Logic in Computer Science*, pages 386–395. IEEE Computer Society Press, June 1989.

[15] Zhaohui Luo, Robert Pollack, and Paul Taylor. How to use LEGO. Technical Report LFCS–TN–27, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[16] David MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 277–286. ACM SIGPLAN/SIGACT, 1986.

[17] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[18] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):57–77, January 1989.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[20] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.

[21] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, Computer Laboratory, University of Cambridge, January 1990.

[22] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. Pacific Grove, California, June 1989. IEEE Computer Society Press.

[23] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[24] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[25] Randy Pollack. Implicit syntax. In G. Huet and G. Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 421–434. Preliminary Version, May 1990.

[26] David Pym. *Proofs, Search and Computation in General Logic.* PhD thesis, University of Edinburgh, 1990. Available as CST-69-90, also published as ECS-LFCS-90-125.

[27] Anne Salvesen. The Church-Rosser theorem for LF with $\beta\eta$-reduction. Unpublished notes to a talk given at the First Workshop on Logical Frameworks in Antibes, May 1990.

[28] D. Sannella and R. Burstall. Structured theories in LCF. Technical Report CSR-129-83, University of Edinburgh, 1983.

[29] D. T. Sannella and L. A. Wallen. A calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, 12:147–177, 1992. A preliminary version appears in the Proceedings of the 4th Symposium on Logic Programming, San Francisco, September 1987.

[30] Matthias Weber. *A Meta-Calculus for Formal System Development.* R. Oldenbourg Verlag, München/Wien, 1991.